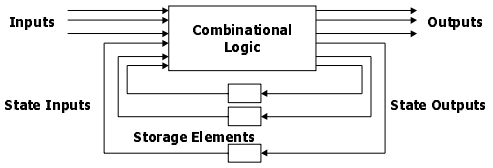
Sequential logic implementation

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Moore and Mealy)
  - representation of memory (states)
  - changes in state (transitions)
- Basic sequential circuits
  - shift registers
  - counters
- Design procedure
  - state diagrams
  - state transition table
  - next state functions

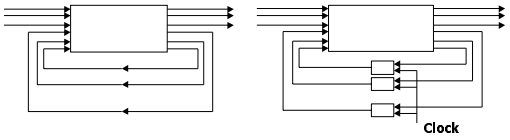
Abstraction of state elements

- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



Forms of sequential logic

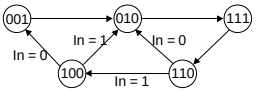
- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)



Finite state machine representations

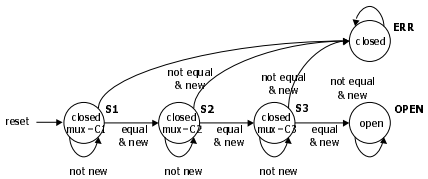
- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements

- Sequential logic
  - sequences through a series of states
  - based on sequence of values on input signals
  - clock period defines elements of sequence



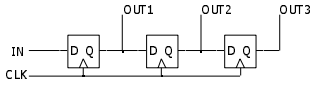
Example finite state machine diagram

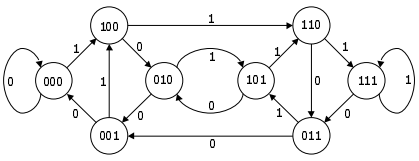
- Combination lock from introduction to course



Can any sequential system be represented with a state diagram?

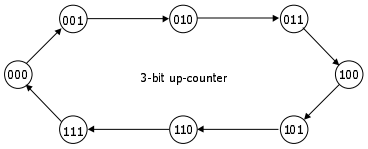
- Shift register
  - input value shown on transition arcs
  - output values shown within state node





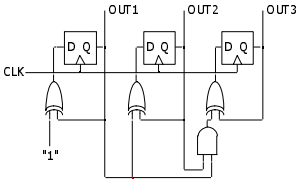
Counters are simple finite state machines

- Counters
  - proceed through well-defined sequence of states in response to enable
- Many types of counters: binary, BCD, Gray-code
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...



How do we turn a state diagram into logic?

- Counter
  - 3 flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value
    - don't wait too long as that is low performance

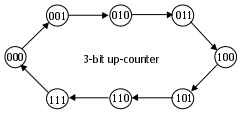


FSM design procedure

- Start with counters
  - simple because output is just state
  - simple because no choice of next state based on input
- State diagram to state transition table
  - tabular form of state diagram
  - like a truth-table
- State encoding
  - decide on representation of states
  - for counters it is simple: just its value
- Implementation
  - flip-flop for each state bit
  - combinational logic based on encoding

FSM design procedure: state diagram to encoded state transition table

- Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters – just use value



current state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

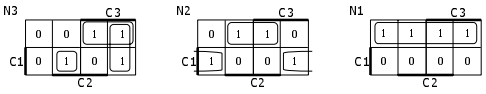
Implementation

- D flip-flop for each state bit
- Combinational logic based on encoding

C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

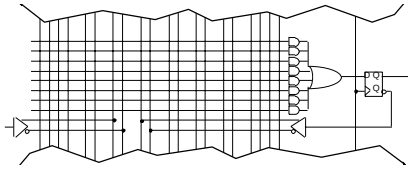
notation to show function represent input to D-FF

N1 := C1'
N2 := C1C2' + C1'C2
  := C1 xor C2
N3 := C1C2C3' + C1'C3 + C2'C3
  := C1C2C3' + (C1' + C2')C3
  := (C1C2) xor C3



Implementation (cont'd)

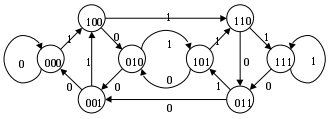
- Programmable logic building block for sequential logic
  - macro-cell: FF + logic
    - D-FF
    - two-level logic capability like PAL (e.g., 8 product terms)



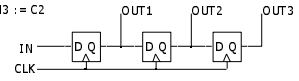
Another example

- Shift register
  - input determines next state

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

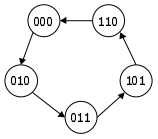


N1 := In
N2 := C1
N3 := C2



More complex counter example

- Complex counter
  - repeats 5 states in sequence
  - not a binary number representation
- Step 1: derive the state transition diagram
  - count sequence: 000, 010, 011, 101, 110
- Step 2: derive the state transition table from the state transition diagram

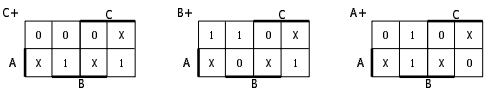


Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

note the don't care conditions that arise from the unused state codes

More complex counter example (cont'd)

- Step 3: K-maps for next state functions



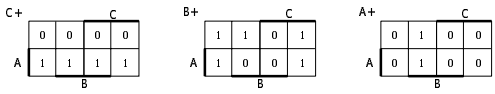
C+ := A

B+ := B' + AC'

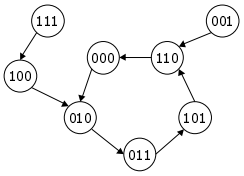
A+ := BC'

Self-starting counters (cont'd)

- Re-deriving state transition table from don't care assignment

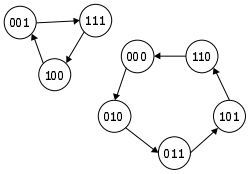


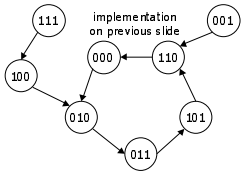
Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



Self-starting counters

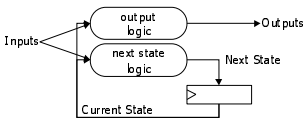
- Start-up states
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design counter so that invalid states eventually transition to a valid state
  - may limit exploitation of don't cares





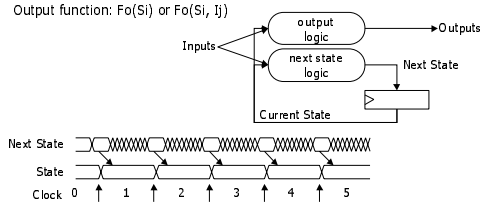
State machine model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)



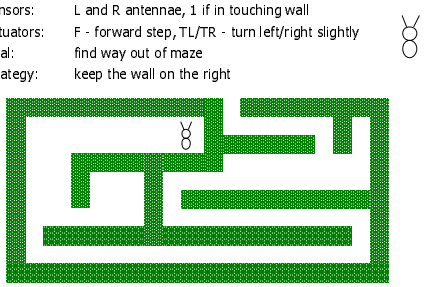
State machine model (cont'd)

- States: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: Fs(Si, Ij)
- Output function: Fo(Si) or Fo(Si, Ij)

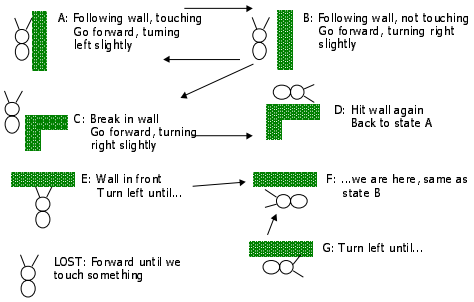


Example: ant brain (Ward, MIT)

- Sensors: L and R antennae, 1 if in touching wall
- Actuators: F - forward step, TL/TR - turn left/right slightly
- Goal: find way out of maze
- Strategy: keep the wall on the right

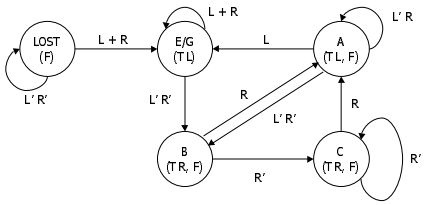


Ant behavior



Designing an ant brain

- State diagram

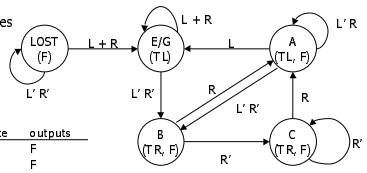


Synthesizing the ant brain circuit

- Encode states using a set of state variables
  - arbitrary choice - may affect cost, speed
- Use transition truth table
  - define next state function for each state variable
  - define output function for each output
- Implement next state and output functions using combinational logic
  - 2-level logic (ROM/PLA/PAL)
  - multi-level logic
  - next state and output functions can be optimized together

Transition truth table

- Using symbolic states and outputs



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	–	1	E/G	F
LOST	1	–	E/G	F
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	–	E/G	TL, F
B	–	0	C	TR, F
B	–	1	A	TR, F
...	...	...	...	...

Synthesis

- 5 states : at least 3 state variables required (X, Y, Z)
  - state assignment (in this case, arbitrarily chosen)

LOST - 000
E/G - 001
A - 010
B - 011
C - 100

state X,Y,Z	L R	next state X', Y', Z'	outputs F TR TL
0 0 0	0 0	0 0 0	1 0 0
0 0 0	0 1	0 0 1	1 0 0
...	... ...	...	...
0 1 0	0 0	0 1 1	1 0 1
0 1 0	0 1	0 1 0	1 0 1
0 1 0	1 0	0 0 1	1 0 1
0 1 0	1 1	0 0 1	1 0 1
0 1 1	0 0	1 0 0	1 1 0
0 1 1	0 1	0 1 0	1 1 0
...	... ...	...	...

it now remains to synthesize these 6 functions

Synthesis of next state and output functions

state X,Y,Z	inputs L R	next state X+,Y+,Z+	outputs F TR TL
0 0 0	0 0	0 0 0	1 0 0
0 0 0	- 1	0 0 1	1 0 0
0 0 0	1 -	0 0 1	1 0 0
0 0 1	0 0	0 1 1	0 0 1
0 0 1	- 1	0 1 0	0 0 1
0 0 1	1 -	0 1 0	0 0 1
0 1 0	0 0	0 1 1	1 0 1
0 1 0	0 1	0 1 0	1 0 1
0 1 0	1 -	0 0 1	1 0 1
0 1 1	- 0	1 0 0	1 1 0
0 1 1	- 1	0 1 0	1 1 0
1 0 0	- 0	1 0 0	1 1 0
1 0 0	- 1	0 1 0	1 1 0

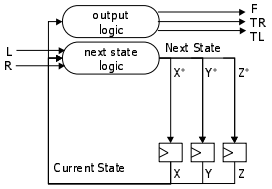
e.g.

$TR = X + Y Z$

$X^+ = X R' + Y Z R' = R' TR$

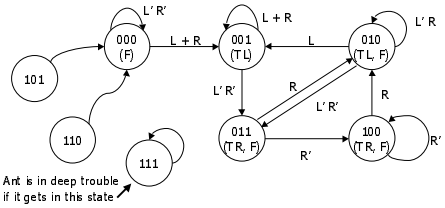
Circuit implementation

- Outputs are a function of the current state only - Moore machine



Don't cares in FSM synthesis

- What happens to the "unused" states (101, 110, 111)?
- They were exploited as don't cares to minimize the logic
  - if the states can't happen, then we don't care what the functions do
  - if states do happen, we may be in trouble

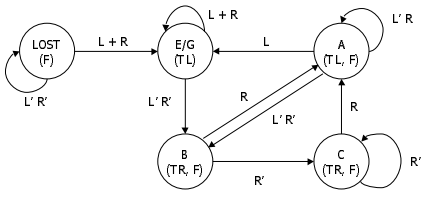


State minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state are equivalent if they are impossible to distinguish from the outputs of the FSM, i. e., for any input sequence the outputs are the same
- Two conditions for two states to be equivalent:
  - 1) output must be the same in both states
  - 2) must transition to equivalent states for all input combinations

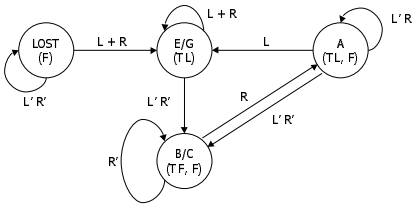
Ant brain revisited

- Any equivalent states?



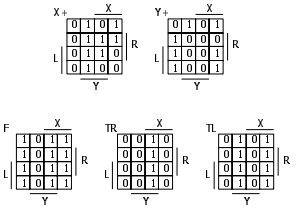
New improved brain

- Merge equivalent B and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3



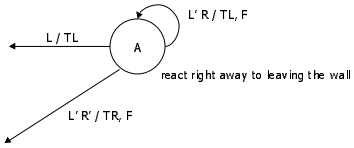
New brain implementation

state X,Y	inputs L	R	next state X',Y'		outputs F	TR	TL
0 0	0	0	0	0	1	0	0
0 0	-	1	0	1	1	0	0
0 0	1	-	0	1	1	0	0
0 1	0	0	1	1	0	0	1
0 1	-	1	1	0	0	0	1
0 1	1	-	1	0	0	0	1
1 0	0	0	1	1	1	0	1
1 0	0	1	1	0	1	0	1
1 0	1	-	0	1	1	0	1
1 1	-	0	0	0	1	1	0
1 1	-	1	1	0	1	1	0



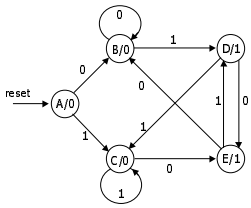
Mealy vs. Moore machines

- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our ant brain is a Moore machine
  - output does not react immediately to input change
- We could have specified a Mealy FSM
  - outputs have immediate reaction to inputs
  - as inputs change, so does next state, doesn't commit until clocking event



Specifying outputs for a Moore machine

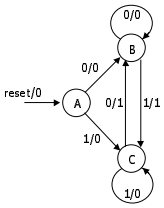
- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



reset	input	current state	next state	output
1	–	–	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Specifying outputs for a Mealy machine

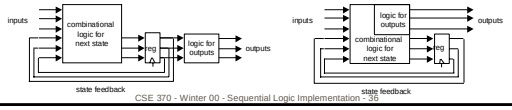
- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10



reset	input	current state	next state	output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

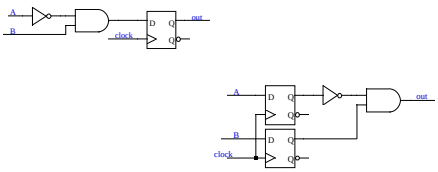
Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - different outputs on arcs (n^2) rather than states (n)
- Moore machines are safer to use
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback
- Mealy machines react faster to inputs
  - react in same cycle – don't need to wait for clock
  - in Moore machines, more logic may be necessary to decode state into outputs – more gate delays after



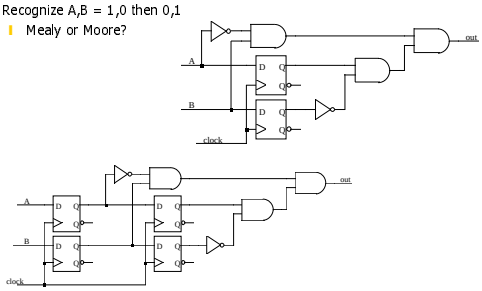
Mealy and Moore examples

- Recognize A,B = 0,1
  - Mealy or Moore?



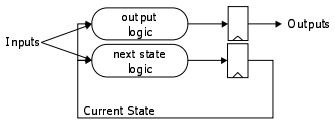
Mealy and Moore examples (cont'd)

- Recognize A,B = 1,0 then 0,1
  - Mealy or Moore?



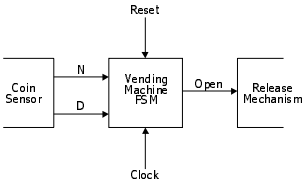
Registered Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - registered state AND outputs
  - avoids 'glitchy' outputs
  - easy to implement in PLDs
- Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering
  - view outputs as expanded state vector



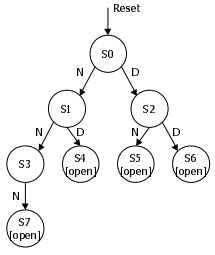
Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change



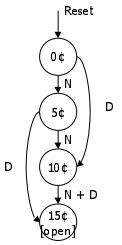
Example: vending machine (cont'd)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



Example: vending machine (cont'd)

- Minimize number of states - reuse states whenever possible



present state	inputs D	inputs N	next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	–	–
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	–	–
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	–	–
15¢	–	–	15¢	1

symbolic state table

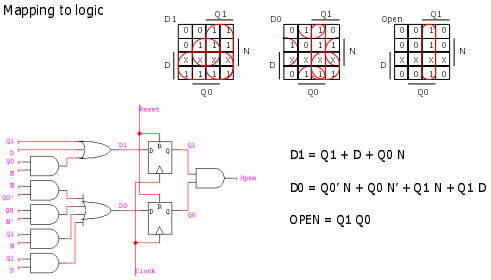
Example: vending machine (cont'd)

- Uniquely encode states

present state Q1 Q0	inputs D' N	next state D1 D0	output open
0 0	0 0	0 0	0
	0 1	0 1	0
	1 0	1 0	0
	1 1	– –	–
0 1	0 0	0 1	0
	0 1	1 0	0
	1 0	1 1	0
	1 1	– –	–
1 0	0 0	1 0	0
	0 1	1 1	0
	1 0	1 1	0
	1 1	– –	–
1 1	– –	1 1	1

Example: vending machine (cont'd)

- Mapping to logic



D1 = Q1 + D + Q0 N

D0 = Q0' N + Q0 N' + Q1 N + Q1 D

OPEN = Q1 Q0

Example: vending machine (cont'd)

- One-hot encoding

present state Q3 Q2 Q1 Q0	inputs D N	next state output D3 D2 D1 D0	open
0 0 0 1	0 0	0 0 0 1	0
	0 1	0 0 1 0	0
	1 0	0 1 0 0	0
	1 1	- - - -	-
0 0 1 0	0 0	0 0 1 0	0
	0 1	0 1 0 0	0
	1 0	1 0 0 0	0
	1 1	- - - -	-
0 1 0 0	0 0	0 1 0 0	0
	0 1	1 0 0 0	0
	1 0	1 0 0 0	0
	1 1	- - - -	-
1 0 0 0	- -	1 0 0 0	1

D0 = Q0 D' N'

D1 = Q0 N + Q1 D' N'

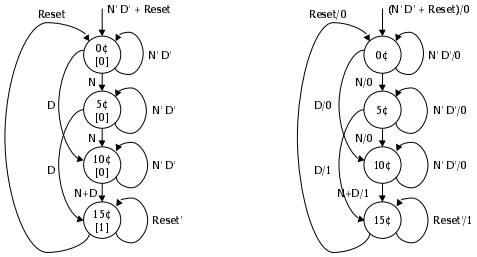
D2 = Q0 D + Q1 N + Q2 D' N'

D3 = Q1 D + Q2 D + Q2 N + Q3

OPEN = Q3

Equivalent Mealy and Moore state diagrams

- Moore machine
  - outputs associated with state
- Mealy machine
  - outputs associated with transitions

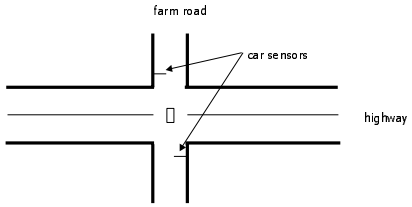


Example: traffic light controller

- A busy highway is intersected by a little used farmroad
- Detectors C sense the presence of cars waiting on the farmroad
  - with no car on farmroad, light remain green in highway direction
  - if vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green
  - these stay green only as long as a farmroad car is detected but never longer than a set interval
  - when these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green
  - even if farmroad vehicles are waiting, highway gets at least a set interval as green
- Assume you have an interval timer that generates:
  - a short time pulse (TS) and
  - a long time pulse (TL),
  - in response to a set (ST) signal.
  - TS is to be used for timing yellow lights and TL for green lights

Example: traffic light controller (cont')

- Highway/farm road intersection



Example: traffic light controller (cont')

- Tabulation of inputs and outputs

inputs	description	outputs	description
reset	place FSM in initial state	HG, HY, HR	assert green/yellow/red highway lights
C	detect vehicle on the farm road	FG, FY, FR	assert green/yellow/red highway lights
TS	short time interval expired	ST	start timing a short or long interval
TL	long time interval expired		

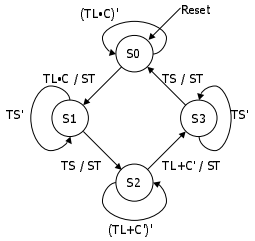
- Tabulation of unique states – some light configurations imply others

state	description
S0	highway green (farm road red)
S1	highway yellow (farm road red)
S2	farm road green (highway red)
S3	farm road yellow (highway red)

Example: traffic light controller (cont')

- State diagram

S0: HG
S1: HY
S2: FG
S3: FY



Example: traffic light controller (cont')

- Generate state table with symbolic states
- Consider state assignments

output encoding – similar problem to state assignment (Green – 00, Yellow – 01, Red – 10)

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	Green	Red
–	0	–	HG	HG	0	Green	Red
1	1	–	HG	HY	1	Green	Red
–	–	0	HY	HY	0	Yellow	Red
–	–	1	HY	FG	1	Yellow	Red
1	0	–	FG	FG	0	Red	Green
0	–	–	FG	FY	1	Red	Green
–	1	–	FG	FY	1	Red	Green
–	–	0	FY	FY	0	Red	Yellow
–	–	1	FY	HG	1	Red	Yellow

SA1:	HG – 00	HY – 01	FG – 11	FY – 10	
SA2:	HG – 00	HY – 10	FG – 01	FY – 11	
SA3:	HG – 0001	HY – 0010	FG – 0100	FY – 1000	(one-hot)

Logic for different state assignments

- SA1

NS1 = C•TL'•PS1•PS0 + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
NS0 = C•TL•PS1'•PS0' + C•TL'•PS1•PS0 + PS1'•PS0

ST = C•TL•PS1'•PS0' + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
H1 = PS1 H0 = PS1'•PS0
F1 = PS1' F0 = PS1•PS0'

- SA2

NS1 = C•TL•PS1' + TS'•PS1 + C'•PS1'•PS0
NS0 = TS•PS1•PS0' + PS1'•PS0 + TS'•PS1•PS0

ST = C•TL•PS1' + C'•PS1'•PS0 + TS•PS1
H1 = PS0 H0 = PS1•PS0'
F1 = PS0' F0 = PS1•PS0

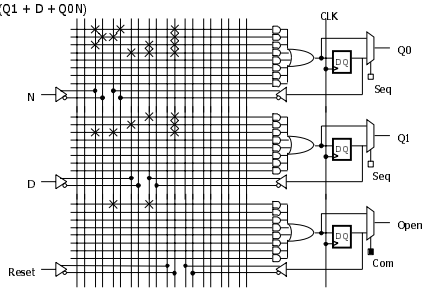
- SA3

NS3 = C'•PS2 + TL•PS2 + TS'•PS3 NS2 = TS•PS1 + C•TL'•PS2
NS1 = C•TL•PS0 + TS'•PS1 NS0 = C'•PS0 + TL'•PS0 + TS•PS3

ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3
H1 = PS3 + PS2 H0 = PS1
F1 = PS1 + PS0 F0 = PS3

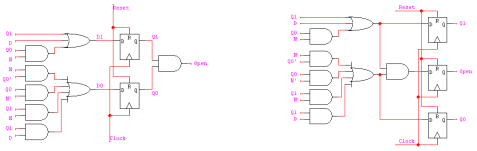
Vending machine example (PLD mapping)

D0 = reset'(Q0'N + Q0N' + Q1N + Q1D)
D1 = reset'(Q1 + D + Q0N)
OPEN = Q1Q0



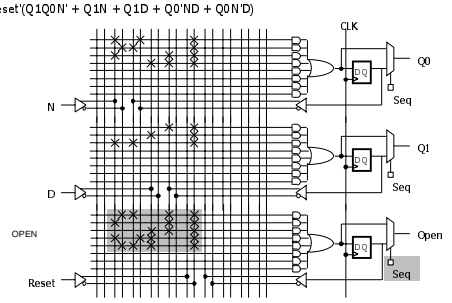
Vending machine (cont'd)

- OPEN= Q1Q0 creates a combinational delay after Q1 and Q0 change
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay
- OPEN= reset'(Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
  = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)
- Implementation now looks like a synchronous Mealy machine
  - it is common for programmable devices to have FF at end of logic



Vending machine (retimed PLD mapping)

OPEN = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)



Finite state machine optimization

- State minimization
  - fewer states require fewer state bits
  - fewer bits require fewer logic equations
- Encodings: state, inputs, outputs
  - state encoding with fewer bits has fewer equations to implement
    - however, each may be more complex
  - state encoding with more bits (e.g., one-hot) has simpler equations
    - complexity directly related to complexity of state diagram
  - input/output encoding may or may not be under designer control

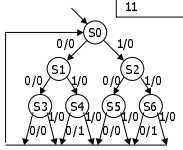
Algorithmic approach to state minimization

- Goal – identify and combine states that have equivalent behavior
- Equivalent states:
  - same output
  - for all input combinations, states transition to same or equivalent states
- Algorithm sketch
  - 1. place all states in one set
  - 2. initially partition set based on output behavior
  - 3. successively partition resulting subsets based on next state transitions
  - 4. repeat (3) until no further partitioning is required
    - states left in the same set are equivalent
  - polynomial time procedure

State minimization example

- Sequence detector for 010 or 110

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0



Method of successive partitions

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0

(S0 S1 S2 S3 S4 S5 S6)

(S0 S1 S2 S3 S5) (S4 S6)

(S0 S3 S5) (S1 S2) (S4 S6)

(S0) (S3 S5) (S1 S2) (S4 S6)

S1 is equivalent to S2

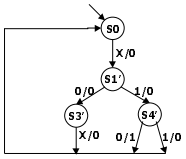
S3 is equivalent to S5

S4 is equivalent to S6

Minimized FSM

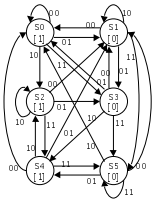
- State minimized sequence detector for 010 or 110

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0



More complex state minimization

- Multiple input example



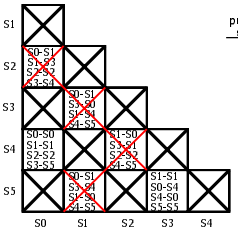
inputs here

present state	next state 00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

symbolic state transition table

Minimized FSM

- Implication chart method
  - cross out incompatible states based on outputs
  - then cross out more cells if indexed chart entries are already crossed out



present state	next state 00	01	10	11	output
S0'	S0'	S1	S2	S3'	1
S1	S0'	S3'	S1	S3'	0
S2	S1	S3'	S2	S0'	1
S3'	S1	S0'	S0'	S3'	0

minimized state table
(S0==S4) (S3==S5)

Minimizing incompletely specified FSMs

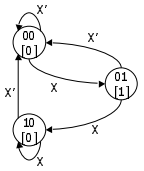
- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present

e.g.,	state	output	
	S0	– 0	S1 is compatible with both S0 and S2
	S1	1 –	but S0 and S2 are incompatible
	S2	– 1	

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing states may not yield best circuit

- Example: edge detector - outputs 1 when last two input changes from 0 to 1



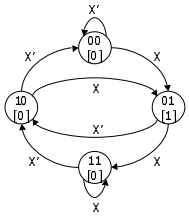
X	Q_1	Q_0	Q_1^*	Q_0^*
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
–	1	1	0	0

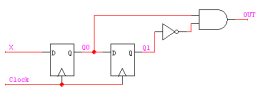
$Q_1^* = X\ (Q_1 \text{ xor } Q_0)$

$Q_0^* = X\ Q_1 Q_0$

Another implementation of edge detector

- "Ad hoc" solution - not minimal but cheap and fast





State assignment

- Choose bit vectors to assign to each "symbolic" state
  - with n state bits for m states there are $2^n! / (2^n - m)!$
    [log n <= m <= 2^n]
  - 2^n codes possible for 1st state, 2^n–1 for 2nd, 2^n–2 for 3rd, ...
  - huge number even for small values of n and m
    - intractable for state machines of any size
    - heuristics are necessary for practical solutions
  - optimize some metric for the combinational logic
    - size (amount of logic and number of FFs)
    - speed (depth of logic and fanout)
    - dependencies (decomposition)

State assignment strategies

- Possible strategies
  - sequential – just number states as they appear in the state table
  - random – pick random codes
  - one-hot – use as many state bits as there are states (bit=1 –> state)
  - output – use outputs to help encode states
  - heuristic – rules of thumb that seem to work in most cases
- No guarantee of optimality – another intractable problem

One-hot state assignment

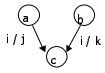
- Simple
  - easy to encode
  - easy to debug
- Small logic functions
  - each state function requires only predecessor state bits as input
- Good for programmable devices
  - lots of flip-flops readily available
  - simple functions with small support (signals its dependent upon)
- Impractical for large machines
  - too many states require too many flip-flops
  - decompose FSMs into smaller pieces that can be one-hot encoded
- Many slight variations to one-hot
  - one-hot + all-0

Heuristics for state assignment

- Adjacent codes to states that share a common next state
  - group 1's in next state map

I	Q	Q+	O
i	a	c	j
i	b	c	k

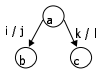
c = i * a + i * b



- Adjacent codes to states that share a common ancestor state
  - group 1's in next state map

I	Q	Q+	O
i	a	b	j
k	a	c	l

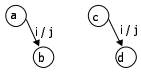
b = i * a
c = k * a



- Adjacent codes to states that have a common output behavior
  - group 1's in output map

I	Q	Q+	O
i	a	b	j
i	c	d	j

j = i * a + i * c
b = i * a
d = i * c



General approach to heuristic state assignment

- All current methods are variants of this
  - 1) determine which states "attract" each other (weighted pairs)
  - 2) generate constraints on codes (which should be in same cube)
  - 3) place codes on Boolean cube so as to maximize constraints satisfied (weighted sum)
- Different weights make sense depending on whether we are optimizing for two-level or multi-level forms
- Can't consider all possible embeddings of state clusters in Boolean cube
  - heuristics for ordering embedding
  - to prune search for best embedding
  - expand cube (more state bits) to satisfy more constraints

Output-based encoding

- Reuse outputs as state bits - use outputs to help distinguish states
  - why create new functions for state bits when output can serve as well
  - fits in nicely with synchronous Mealy implementations

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	00	10
–	0	–	HG	HG	0	00	10
1	1	–	HG	HY	1	00	10
–	–	0	HY	HY	0	01	10
–	–	1	HY	FG	1	01	10
1	0	–	FG	FG	0	10	00
0	–	–	FG	FY	1	10	00
–	1	–	FG	FY	1	10	00
–	–	0	FY	FY	0	10	01
–	–	1	FY	HG	1	10	01

HG – ST' H1' H0' F1' F0' + ST H1 H0' F1' F0
HY – ST H1' H0' F1' F0' + ST' H1' H0 F1' F0'
FG – ST H1' H0 F1' F0' + ST' H1 H0' F1' F0'
HY – ST H1 H0' F1' F0' + ST' H1 H0 F1' F0

Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

Current state assignment approaches

- For tight encodings using close to the minimum number of state bits
  - best of 10 random seems to be adequate (averages as well as heuristics)
  - heuristic approaches are not even close to optimality
  - used in custom chip design
- One-hot encoding
  - easy for small state machines
  - generates small equations with easy to estimate complexity
  - common in FPGAs and other programmable logic
- Output-based encoding
  - ad hoc - no tools
  - most common approach taken by human designers
  - yields very small circuits for most FSMs

Sequential logic implementation summary

- Models for representing sequential circuits
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- Finite state machine design procedure
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic
- Implementation of sequential logic
  - state minimization
  - state assignment
  - support in programmable logic devices